We present a Domain Specific Language for Indoor Scenes designed to express possible object locations within an indoor scene in semantically understandable terms. We show how our symbolic representation encourages object placements that are diverse and retain the specificity required for realistic scene generation. Based on this new language we also present an autoregressive transformer architecture equipped with a new augmented edge attention mechanism designed to infer programs from 3D scene and object data. We train this model with data generated by a program extraction process that converts 3D scene data into programs. Abstract of “Neurosymbolic Methods for Indoor Scene Synthesis” by Adrian Chang, Sc.B., Brown University, May 2023.
Neurosymbolic Methods for Indoor Scene Synthesis

by

Adrian Chang

Sc.B. Brown University, 2023

A Thesis submitted in partial fulfillment of the requirements for Honors in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2023
This thesis by Adrian Chang is accepted in its present form by the Department of Computer Science as satisfying the research requirement for the awardment of Honors.

Date ______________  

Daniel Ritchie, Reader

Date ______________  

Srinath Sridhar, Reader
Acknowledgements

The author would like to thank Daniel Ritchie for his infectious enthusiasm and insightful feedback, Kai Wang for his mentorship and guidance, as well as Manolis Savva and Angel Chang for their support. This work was completed at the Department of Computer Science at Brown University.
## Contents

1 Introduction  

2 Related Work  
   2.1 Indoor Scene Synthesis  
   2.2 Procedural Modeling, Inverse Procedural Modeling, and Visual Program Induction  

3 A Language for Object Placement  
   3.1 Constraint Execution  
      3.1.1 Location constraints  
      3.1.2 Orientation constraints  

4 Extracting Program Data  
   4.1 Cleaning  
   4.2 Program initialization  
   4.3 Self Training  

5 Learning to Generate Programs  
   5.1 Model Architecture  
   5.2 Object Encoding  
   5.3 Sequence Encoding and Decoding  
   5.4 Edge Attention  

6 Experiments/Results  
   6.1 Examples of Inferred Programs  
      6.1.1 Training Set  
      6.1.2 Validation Set  
   6.2 Experiment in Toy Setting  

7 Limitations and Future Work  

8 Conclusion  
Chapter 1

Introduction

Automatically generating 3D scene data that is both realistic and diverse unlocks a wide range of potential applications. Traditional sources of demand for virtual indoor environments such as interior design, game development, CGI, and VR/AR have only seen increased interest in automatic generation of scene layouts. Scene synthesis decreases the amount of manual labor required for such applications, and controllable scene synthesis unlocks new possible user experiences. In addition to these traditional sources, researchers working on problems such as embodied agent navigation, scene understanding, and scene reconstruction benefit from large-scale data sets of 3D scene data.

Previous methods approach scene modeling and synthesis with procedural modeling techniques that require pre-defined and hand authored rules. Creating such rules is difficult, time-consuming, and requires the skills of experienced artists and designers. Scenes generated by these hand authored rules also often lack diversity due to the limitations of hand-authored scene constraints. Another more recent line work leverages the power of deep generative models to learn these rules implicitly from large-scale datasets. These methods use CNN and Transformer based architectures to auto regressively place objects in a room, i.e. place objects one at a time. Deep generative models can easily learn varied and complex rules for scene generation without the difficulty of human authorship, but these rules lack the interpretability of explicit representations and their semantic correctness. More specifically, these models are difficult to debug due to their parameterization as a black box neural network and they struggle to learn relationships between objects aligned with human perception. Their suggested object placements are often either invalid or underspecified.

In this work, we approach scene modeling from a neuro symbolic perspective, and design a deep generative model that writes programs. This formulation captures the strengths of both approaches. The deep generative model learns complex and varied placement rules from the dataset automatically, but expresses them in an interpretable explicit representation. The language design leverages human intuition about scene construction and expresses relationships between objects of an indoor scene in terms of how humans use them. We show that this injection of human knowledge into the model results in suggested object placements that are semantically valid.

In line with these goals and motivations, we introduce a language designed to express object
placements within a scene in terms of functional constraints. These programs take as input a scene defined by its intrinsic geometry (e.g. walls, floor plan, doors), whatever objects currently reside within the scene, as well as a query object to be placed within the scene. The program outputs a binary mask over the top down orthographic view of the scene. Each element of the binary mask represents a possible centroid location of the query object. Figure 3.1 shows an example of a program’s inputs, execution, and outputs.

In order to automatically generate programs from a given scene and object we design a neural network to write programs and train it on program data extracted from available scene data in the 3D-FRONT dataset. Inspired by recent advances in autoregressive sequence modeling with the transformer architecture, we treat the problem as a sequence-to-sequence translation task where the input sequence are the objects currently in the scene and the target sequence is the program as a sequence. We find that the ordinary attention mechanism fails to correctly attend over the scene objects, so we introduce a new Edge Attention Mechanism which allows the transformer model to attend over both per object information and inter-object information such as distance and directional relationships. Figure 5.1 shows the full model architecture and Figure 5.3 describes the Edge Attention Mechanism.

Paired scene and program data is unavailable, and the available scene data contains no annotations for what the ground truth object distributions are. Direct supervised learning is thus infeasible as the “ground truth” programs are unknown, but it is still possible to obtain high quality paired scene program data with an iterative self-training approach. Our initial training data are programs that only place objects where they are originally found. We then propose leveraging the generative model to refine these programs.

To summarize, our contributions are as follows

- A domain specific language (DSL) for indoor scenes and its program executor which generate masks of possible object placements
- A program extraction process based on geometric heuristics, data cleaning, and an iterative self training approach designed to discover the “ground truth” programs
- An autoregressive transformer model equipped with a new edge attention mechanism which learns to generate programs from a given scene and object. We treat extracted programs labels as training data for this network.
Chapter 2

Related Work

This section describes related work in Indoor Scene Synthesis, Procedural Modeling, and Visual Program Induction

2.1 Indoor Scene Synthesis

Early traditional work in 3D scene generation formulated the task as an optimization procedure guided either by cost functions derived from interior design principles [6] or pairwise statistical relationships between objects [21]. Other proposed methods use human activity and usage of the indoor scene as a guiding principle. This can take the form of spatial optimization schemes based on workplace efficiency metrics [22], human-centric stochastic grammars [11], or interaction maps of human activity [2].

3D deep learning unlocks end to end differentiable frameworks capable of processing large datasets of scenes. Many of these works are autoregressive scene generation models that place objects into a room one at a time until the room is fully furnished. Richie et al. [12, 19] uses CNNs operating over a top-down image representation of the scene to predict distributions of object placements. Other works use transformers to model object attributes with either fixed [20] or arbitrary [8] object orderings. Other works represent scenes as graphs and use either GNNs to generate scenes [18], or transformers to predict constraint graphs [7]. Most recently diffusion models operating over scene graphs have shown promising results [15].

2.2 Procedural Modeling, Inverse Procedural Modeling, and Visual Program Induction

Related to this work are procedural modeling systems for computer graphics such as L systems [10] or Constructive Solid Geometry [13]. Most procedural modeling systems use grammars to define a program tree. Our domain specific language takes on a similar structure, and aims to describe the
structure of 3D scenes in an explicit format. Other domain specific languages such as ShapeAssembly [4] and Geocode [9] aim to describe the structure of 3D shapes in a discrete interpretable manner. ShapeAssembly also introduces methods for generating programs, which in turn generate shapes, automatically. We also introduce methods for writing programs automatically.

Inverse procedural modeling, or learning a procedural model from a set of examples, is also related to this work. These methods aim to generate interpretable stochastic programs that faithfully reproduce a given distribution of designs and generate novel examples in line with the exemplar design patterns. One such work uses probabilistic context free grammars (PCFGs) to model hierarchical designs in a grammar induction pipeline [14]. The optimization begins with the “most-specific” grammars and converges to a grammar which is not too specific and not too general. Similar to this work, we also start by inferring specific, non-general programs, and propose methods to produce more general programs.

Another related line of work is visual program induction, or the process of inferring programs from single entities [1]. The PLAD (Pseudo-Labels and Approximate Distributions) framework is a conceptual framework which groups techniques of program inference under a single framework [5]. Our proposed self-training approach falls under the PLAD framework, where the programs are “pseudo-labels” and the output mask is the “approximate distribution.”
A Language for Object Placement

Figure 3.1: Example of Program Execution: This program takes in a scene containing one bed (the gray rectangle) and a nightstand to the right of the bed (yellow rectangle). The query object is another nightstand. The subprogram on the right subtree puts the nightstand within arm’s reach of the bed’s left side. The location constraint \texttt{reachable_by_arm} predicts locations on the left side of the bed for all possible orientations of the nightstand (represented by the black rectangles). The align constraint ensures that the nightstand points in the same direction of the bed which involves rotating the nightstand by 0 degrees in this case. As such the top left mask is completely black, as any location with orientation 0 satisfies that constraint. The left subtree attaches the nightstand to the same wall as the bed and ensures it points in the same direction. In total the output of the program states that the nightstand must be within arm’s reach of the bed’s left side, attached to the same wall, and point in the same direction. The predicted mask also matches the ground truth placement.
Our goal is to design a language capable of expressing a wide range of possible placements in semantically meaningful terms. We chose to base this language on the structure of CSG trees, but instead of operating over a continuous 3D space, the programs operate over binary masks. Leaf nodes of this tree are functional constraints that capture certain kinds of human activity and object function. When executed these constraints produce binary masks that the other nodes operate over.

Binary masks are discretized among 3 dimensions. The three dimensions represent the width of the room, the height of the room, and the possible orientations of the object to place respectively. This third dimension of the mask is necessary because the validity of an object’s centroid position is dependent on its orientation. The orientation of an object dictates the centroid locations the program executor predicts. We represent the orientation of an object as its rotation about the up axis of the room and discretize this angular domain into 4 equal size bins.

3.1 Constraint Execution

We define the following functional constraints and argument types. Functional constraints fall under two categories. Location constraints assume nothing about an object’s orientation and predict an object’s possible centroid locations for every possible orientation. Orientation constraints constrain the possible orientations of the object within a scene in addition to the location.

Every object comes with a label of its semantic front as well as whether or not it holds humans. We make the simplification that all objects have only one semantic front, although this is not true for many objects. Objects and scenes in the 3D-FRONT dataset also come with semantically meaningful sizes, scales, and distances. As such the geometric heuristics described for each constraint are based on physically meaningful quantities such as the average reaching distance. 5 total directions are specified in the language \{(Up, Down, Left, Right, Null)\} and all directions are specified within the local coordinate frame of the reference object. The Null direction exists because orientation constraints do not need a direction specification.

3.1.1 Location constraints

- **attach(query object, reference object, direction)**: Output the possible centroid locations of the query object such that it is within 15 centimeters of the reference object in the direction specified.

- **reachable_by_arm(query object reference object, direction)**: Output the possible centroid locations of the query object such that it is between 15-60 centimeters of the reference object in the direction specified. The reference object must also hold humans (i.e. bed, chair).

3.1.2 Orientation constraints

- **align(query object, reference object)**: Constrain the possible orientation of the query object such that its semantic front points in the same direction as the reference object’s
semantic front.

- **face(query object, reference object)**: Constrain the possible locations of the query object such that its semantic front points toward the reference object. Evaluate this for every possible orientation.

Executing a program involves executing each constraint in the tree and then combining the masks accordingly. We apply a final post-processing step that removes placements of the query object which intersect with other objects in the scene beyond a specified threshold. Figure 3.1 walks through an example program.
Chapter 4

Extracting Program Data

Programs give us a symbolic representation capable of expressing distributions of object placements. We want to design a procedure which discovers which programs most accurately model a given dataset of scenes. A new dataset of “ground truth” programs enables the generation of new scenes and is itself an interesting subject of study. We first clean available scene data, extract the most restrictive programs possible from these scenes, and then use those programs to initialize a self training process.

4.1 Cleaning

We find that around 20% of scenes in the 3D-FRONT dataset [3] have significant interpenetration between scene objects or large portions of objects lying outside the scene. Inferring inter-object relationships such as which side an object is on in relation to another object becomes difficult when the objects intersect. Certain geometric quantities such as what percent of a source object’s side does a target object overlap with can ameliorate situations of shallow penetration, but scenes with major inter-object collisions are difficult to accurately predict. Such scenes are removed from the dataset.

4.2 Program initialization

We want to extract programs which produce semantically valid object placements. A procedure which produces programs that describe all possible valid object placements in a scene is difficult to author. and if such a procedure exists, our method is unnecessary. It is simple, however, to find programs that represent a subset of all possible valid placements. For an object in a scene, we apply every possible functional constraint on it so that the final program will only place the object where it was originally found. We call these programs the most restrictive programs.

Previous work has shown that training autoregressive transformer models in a permutation invariant fashion yields results superior to methods that do not [8]. In this same philosophy, we
subsample scenes into scene object pairs that represent every possible combination of objects in the room. Objects that constitute the scene’s intrinsic geometry such as walls or doors are left out of this subsampling. The most restrictive programs extracted from these subsampled scene object pairs constitute the initial training data.

It is possible for these most restrictive programs to become over constrained and produce null masks. The procedure for producing these programs involves “and-ing” every possible constraint together into one CSG tree. In certain cases two branches of the tree that represent two separate distributions of object placements may not overlap, so when combined produce a null mask. Program execution is also sensitive to hyperparameters which affect whether two very close mask distributions overlap. Additionally, these most restrictive programs can contain “extraneous” constraints: constraints which do not affect the final distribution but are present in the program. Null programs are removed from the training set, and extraneous constraints are greedily removed from the remaining programs.

It is also worth mentioning that the opposite case of under constrained programs can also occur. In some cases it is impossible to extract a program of desired specificity. For example, if the room contains only a single bed attached to a wall, the produced program will describe all placements of that bed against the wall, but not a specific placement.

4.3 Self Training

These most restrictive programs produce specific and semantically valid object placements, but lack diversity. They contain no ‘or’ tokens and it is unclear how different object placements outside of their original location might be introduced to the “ground truth” programs. Our proposed solution is a bootstrapped self training process. We train a generative model described in the next section on the most restrictive programs, use this model to infer candidate programs, and then combine current “ground truth” programs with the new proposed programs. A recognition model such as a real fake classifier that determines whether a candidate program is in distribution might aid in filtering out invalid programs from the training data. Developing this self training process is a fruitful direction that we leave for future developments of this project.
Chapter 5

Learning to Generate Programs

Figure 5.1: We predict programs in a two pass approach. The first encoder decoder pair predicts the structure sequence and the second encoder decoder pair predicts the constraint attributes of the program.

Given the programs extracted from the dataset, we now have the data required to train a generative model to write new programs. This section describes the model architecture, input and output representations, and edge attention mechanism used to train this generative model.

5.1 Model Architecture

We treat program generation as a seq-2-seq translation task. The input or source sequence are the objects in the room. The query object, or object to place, is always the last object in this sequence. The output or target sequence is the program. We represent programs as two separate sequences. One sequence represents the topology or structure of the program tree, and the other represents the attributes of each constraint in the program. Each constraint takes the form (constraint type, query object index, reference object index, direction) or \((c_j, q_j, r_j, d_j)\). Constraint attributes are
Figure 5.2: Shown is the sequence representation of a program which the transformer model learns to generate. In this example the structure sequence might be more easily read as [and [and c c] c] and the constraint sequence as [[0, 7, 0, 1], [2, 7, 0, 4], [0, 7, 1, 1]] concatenated into a sequence following inorder traversal (left subtree to root to right subtree). Figure 5.2 shows an example of this conversion to sequence representation. Our model architecture mirrors this program representation.

We train two transformer [16] encoder-decoder pairs in a two pass approach. The first encoder-decoder pair takes in object encodings and outputs the structure or topology of the target program. The second encoder-decoder pair takes as input the source and target sequences of the first pass concatenated together and outputs the attributes of all the program’s constraints as a flattened list. Figure 5.1 shows the network architecture.

5.2 Object Encoding

Objects are represented by their bounding box with attributes category, size, position, orientation, and whether it holds humans \( o_i = \{ t_i, s_i, p_i, o_i, h_i \} \). The category \( t_i \) is an integer id over the total number of object categories in the dataset. \( s_i, p_i \in R^2 \) (height of objects are not considered and all objects are considered grounded). The orientation \( o_i \in R \) of the object is the rotation of the object about the up vector. Holds_humans \( h_i \in [0, 1] \) is a binary flag of whether the object’s purpose is to hold a human.

The object encoder encodes object bounding boxes into an embedding vector \( \in R^d \). A learned embedding of the object category is concatenated to the raw values of the other attributes and passed through an MLP. We opt for this simple approach rather than a conversion of size, position,
and orientation attributes to fourier features similar to ATISS. We find that the object encoding with fourier features causes the model to overfit to specific object sizes and positions.

Instead of encoding the floor plan, and implicitly the walls, as a single object, we encode each wall segment as its own object. This is due to the difficulty of encoding all the pertinent information about a floor plan into a single feature vector, and giving the model access to that information in a salient manner. The model struggles to infer algebraic quantities such as object to wall distances, especially for floor plans with non-convex geometry. These quantities are required to accurately model object placements.

5.3 Sequence Encoding and Decoding

We encode the program’s “structure” sequence with per token learnable embeddings. We also encode the constraint attribute sequence with per token learnable embeddings, but only for tokens that represent the constraint type or direction. Tokens which represent the query or reference object index use their respective object embeddings generated by the object encoder.

For tokens of fixed vocabulary length such as program structure, constraint type, and direction, an MLP head is enough to decode their tokens, but when choosing the reference object index the vocabulary is variable length as the number of objects in a scene is variable. To address this, we pass each reference object head through an MLP to form a pointer embedding $v_j \in \mathbb{R}^d[17]$. For a matrix of object embeddings $E$ of size $N \times D$

$$r_j = \arg\max(\text{Softmax}(Ev_j))$$

The dot product of the pointer embedding with the object embeddings forms a probability distribution over the objects. The reference object is the object with the highest probability mass.

5.4 Edge Attention

We find that ordinary attention does not correctly account for spatial relationships between objects, so we augment the attention mechanism of the transformer encoder to introduce inter object relationships to the input signal. In ordinary attention key, query, and value vectors are computed from linear projections of the input. The key and query vectors compute the attention weights used for a final weighted sum of the Value vectors. For self-attention with input object embeddings $X$ and attention output $Z$

$$Z = \text{Softmax}(\frac{W_qX(W_kX)^T}{\sqrt{d_k}})W_vX$$

In addition to these vectors we also introduce edge values that encode information between objects. For each query object, we compute key and value vectors $K'$ and $V'$ with respect to the query object. The attention weights are computed as $QK^T + QK'^T$, where the $QK'^T$ term acts as a correction weight to the original $QK^T$ attention weights. The weighted sum of $V'$ using these
Figure 5.3: Edge attention takes additional edge embeddings alongside the traditional input. For a given query object (object 1 in the diagram), we transform this embedding into key and value vectors similar to normal attention and compute the attention weights using the original query vector $q$, original key value $k$, and new key value $k'$. The weighted sum of these edge values is summed to the normal output of attention. This process is repeated for every object in the sequence.

The intuition behind this attention weight formulation is that object info in the original embedding vector such as type should inform which edges receive more weight. For edge values $X_i$ where object $i$ is the query object

$$Z_i = Z_i + \text{Softmax}(W_q X_i (W_k X_i)^T + W_q X_i (W_{ek} X_i)^T \sqrt{d_k}) W_{ev} X_i$$

Figure 5.3 shows a visual representation of this process. For simplicity, edges only encode directional relationships between the source and target object (i.e. on the left side of the source object), but can easily incorporate other kinds of information such as distance or relative angles with an MLP projection.
Chapter 6

Experiments/Results

6.1 Examples of Inferred Programs

Shown below are examples of ground truth programs and their inferred counterparts. The left column represents inferred programs and the right column represents the “ground truth” or most restrictive programs.

6.1.1 Training Set

Example 1

The query object is a nightstand which appears in the dataset on the right side of the bed. The model predicts placement of the nightstand on the left side of the bed. This is still a valid program,
but differs from the “ground truth.”

**Example 2**

The query object is a bed. The “ground truth” program just attaches and aligns it with a wall in the scene. The predicted program also attempts to place the bed within arm’s reach of the chair’s right side. Placement of the bed on the right side of the chair however would result in a bed outside the bounds of the room, so the program is null.

**Example 3**
The query object is a nightstand which appears in the dataset on the right side of the bed. The predicted program is identical to the “ground truth.”

**Example 4**

The query object is a bed. The predicted program relaxes the constraint that the bed must face one of the wardrobes.

**Example 5**

The query object is a wardrobe that appears in a corner of the scene. The predicted program
predicts a different corner in the scene as placement.

6.1.2 Validation Set

Example 1

The query object is a desk which appears in front of a chair in the dataset. The predicted program places the desk in a nearby corner rather than in front of and facing the chair as in the extracted program.

Example 2
The query object is a bed which appears on the left side of a nightstand. The predicted program is identical.

**Example 3**

The query object is a nightstand which appears on the left side of the bed. The predicted program is almost identical except for the use of the attach constraint rather than the reachable by arm constraint.

**Example 4**
The query object is a nightstand which in the extracted program is only attached to a wall. The predicted program attempts to attach the nightstand to the same wall but also make it face the desk in the room. This is impossible, so the predicted mask is null.

**Example 5**

The query object is a wardrobe. The extracted program attaches it to the bottom wall, but contains extraneous constraints which do not significantly affect the final output. The predicted program produces the same output mask, but with less constraints.
6.2 Experiment in Toy Setting

Figure 6.1: The diagram shows an example of a toy dataset used to justify the edge attention mechanism. The logits shown are the activations for choosing the direction the nightstand appears on in relation to the bed.

It is worth mentioning experiments in a toy setting with and without the edge attention mechanism to justify its existence in the model. We construct a toy dataset containing scenes of a bed attached to the different wall segments. Some of the data points also contain a nightstand attached to either the left or right side of the bed. The programs paired with each of these data points predict placements of a nightstand on the left or right hand side of the bed.

It is impossible to perfectly classify this dataset because the nightstand is equally likely to appear on the left or right hand side of the bed (one to many mapping of scenes $x$ to programs $y$). It is possible however to reach selection accuracy of which side the nightstand should appear on above 50%. The network can clue into the spatial relationships between objects in the room. For example, if a wall appears on the right side of the bed, the entire probability mass of which direction the nightstand should fall on should shift the ‘left’ token.

With ordinary attention, the probability mass does not shift in the expected manner. In some cases, the network predicts nightstand placements that would cause object collisions. After introducing the edge attention mechanism, the probability mass shifts completely to the expected direction consistently. See figure 6.1 for a visual example of this test.
Chapter 7

Limitations and Future Work

Our proposed method has multiple limitations that we leave for future work. Our program representation, while interpretable, cannot express all possible object placements because the representation has discretized a continuous spatial and angular domain. The angular domain discretization to 4 restricts the range of possible object rotations to axis aligned directions. Our model also cannot express complex geometries beyond bounding boxes.

There are important aspects of human activity in indoor scenes that are not represented in the currently defined functional constraints. The walkability of a scene and how an object placement affects that walkability, as well as how much walking space is available between two objects are two concepts missing from the language. Introducing the concept of negative space in conjunction with these walkability constraints may yield more semantically rich and accurate programs. Our generative model’s output is also a mask of valid object placements rather than a full scene. To produce high quality scenes it might be necessary to introduce an additional network which transforms a binary mask into a probability distribution that we can sample.
Chapter 8

Conclusion

We introduce a domain specific language for indoor scenes capable of expressing object placements in semantically meaningful terms. An autoregressive transformer model with a new edge attention mechanism can generate plausible programs given a scene and object to place. We show how to “discover” these programs from scene data and generate them automatically. We also present an idea for how to leverage our generative model to iteratively refine an initially overly restrictive set of programs to represent diverse and realistic placements.
Bibliography


